Another essence of dataflow programming (draft 5)

Ben Denckla

February 1, 2006

Abstract

Uustalu and Vene use comonads to separate the dataflow and non-dataflow aspects of an interpreter for a dataflow language. I show that this separation can be achieved more simply if an applicative functor is used instead of a comonad. Applicative functors have recently been described by McBride and Paterson.

1 Introduction

A definitional interpreter is one whose source code is intended not only to implement but also to define a language. As such it is an executable specification of a language. In particular, an interpreter written in an effectless language such as Haskell can provide an executable denotational semantics. If the defined language has effects, monads can be used to separate the effects from the rest of the semantics [5]. This a widely-accepted way to achieve this separation.

Inspired by the successful use of monads to separate the effectful and effectless aspects of a language, Uustalu and Vene [3] use comonads to separate the dataflow and non-dataflow aspects of a language. I will show that it is simpler to achieve this separation with an applicative functor (AF) [2] than a comonad.

With an AF, the core semantics (that of variables, lambdas, and applications) are the same as in a scalar functional language. Only the semantics of the language's constants look a little different, since they are abstracted to accept any ground types conforming to the AF algebra. This includes scalar as well as stream types.

Thus the AF approach reinforces the intuition that intensional dataflow languages like Lucid, although syntactically and pragmatically distinct from functional languages, can be semantically described as functional languages with stream ground types.

In contrast, in the comonadic approach, the core semantics are similar to, but not the same as a scalar functional language. The differences stem from the expanded notion of environment as a comonad and a different notion of function type.

The AF approach is simpler not only with respect to the core semantics but also with respect to the dataflow-specific semantics: making an instance of an AF that yields dataflow semantics is simpler than doing so with a comonad.

This paper will show that the AF approach is simpler in the ways described above by presenting two interpreters for a language I call *Df*. One interpreter will be based on comonads, and the other on AFs. *Df* is a small dataflow language similar to the original Lucid [1] with the notable exception of the lack of support for multiple dimensions. (It has neither the **latest** keyword nor any way to achieve its functionality.)

Df and its comonadic interpreter are close to the language and interpreter presented by Uustalu and Vene. I have tried to make my changes to their code limited to either improvements or stylistic changes. Since my argument is one about simplicity, I have tried to be fair by putting as much effort into simplifying their code as I have into simplifying mine.

The interpreters both have three parts. The first part is a generic interpreter. It is generic in that it is abstracted over a dominant data type (AF or comonad), requiring an instance of this data type to be useful. The second part is an identity instance of the dominant data type that yields a scalar version of *Df* when used with the generic interpreter. The third part is a dataflow instance of the dominant data type that yields the *Df* language when used with the generic interpreter.

The rest of this document presents the syntax of *Df*, the expected values of some example *Df* expressions, and a listing of the two interpreters followed by some concluding remarks. This document is typeset using the lhs2TeX program so certain Haskell symbols have been "de-ASCIIed." Also, all **module** and **import** statements have been removed to conserve space.

2 Syntax

```
type Var   = String
type Hbao = Integer → Integer → Integer
type Hipr = Integer → Bool
data Tm = V Var          -- variable
    |     L Var Tm       -- λ
    |     A Tm Tm        -- application
    |     Y Tm           -- fixed-point operator
    |     C Constant
    |     Fby Tm Tm      -- dataflow "followed by" operator
    |     Nxt            -- dataflow "next" operator
data Constant = N Integer | T Bool
    |           Bao Hbao  -- +, −, ∗, etc.
    |           Ipr Hipr  -- (≡ 0), (≢ 0), (<0), etc.
    |           Not | If
```

3 Sugar

$$
\begin{array}{ll}
ba & = (A\circ)\circ A \quad \text{-- binary apply} \\
yl & = (Y\circ)\circ L \quad \text{-- fixed point of a } \lambda \\
[add, sub, mul, mod\_] & = map\ (ba\circ C\circ Bao)\ [(+),(-),(*),mod] \\
[eqz, gz, neqz] & = map\ (A\ \circ C\circ Ipr)\ \ [(\equiv 0),(>0),(\not\equiv 0)] \\
[not\_, nxt] & = map\ A\ [C\ Not, Nxt] \\
[zero, one] & = map\ (C\circ N)\ [0,1] \\
[x, y] & = map\ V\ [\texttt{"x"}, \texttt{"y"}] \\
if\_ & = ba\circ (A\ (C\ If))
\end{array}
$$

4 Examples

The non-negative integers are denoted by the expression

$yl\ \texttt{"x"}\ (zero\ `Fby`\ (x\ `add`\ one))$

which pretty-prints to $(fix\ (\lambda x \rightarrow (0\ `fby`\ (x+1))))$
and is expected to have value $[0,1,2,3,4,5,6,7,8,9,10,...$

The Fibonacci series is denoted by the expression

$yl\ \texttt{"x"}\ \$\ zero\ `Fby`\ (x\ `add`\ (one\ `Fby`\ x))$

which pretty-prints to $(fix\ (\lambda x \rightarrow (0\ `fby`\ (x + (1\ `fby`\ x)))))$
and is expected to have value $[0,1,1,2,3,5,8,13,21,34,55,...$

5 Common code

$envEmpty = [\,]$
$envLookup\ x\ e = maybe\ \bot\ id\ (lookup\ x\ e)$
$envUpdate\ x\ v\ e = (x,v) : e$
$s\ f\ a\ e = (f\ e)\ (a\ e)$ -- S-combinator (K handled by *const*)
data $Stream\ a = a :< Stream\ a$

6 Generic comonadic semantics

class $Comonad\ d$ **where**
  $counit :: d\ a \rightarrow a$
  $cobind :: (d\ a \rightarrow b) \rightarrow d\ a \rightarrow d\ b$
class $Zippable\ d$ **where** $zzipWith :: (a \rightarrow b \rightarrow c) \rightarrow d\ a \rightarrow d\ b \rightarrow d\ c$
class $(Comonad\ d, Zippable\ d) \Rightarrow DfComonad\ d$ **where**
  $fbyg :: a \rightarrow d\ a \rightarrow a$
  $nxtg :: d\ a \rightarrow a$
data $Val\ d = I\ \{unI\ :: Integer\}$
  | $B\{unB :: Bool\}$
  | $F\{unF :: (d\ (Val\ d) \rightarrow Val\ d)\}$
type $En\ d = [(Var, Val\ d)]$
$ev :: DfComonad\ d \Rightarrow Tm \rightarrow d\ (En\ d) \rightarrow Val\ d$

```
ev (V x)       = envLookup x ∘ counit
ev (L x a)     = λe → F $ λv → ev a (zzipWith (envUpdate x) v e)
ev (A a b)     = s (unF ∘ (ev a)) (cobind (ev b))
ev (Y a)       = ev (A a (Y a))
ev (C c)       = const (evc c)
ev (Fby a b)   = s (fbyg ∘ (ev a)) (cobind (ev b))
ev Nxt         = const (F nxtg)

evc (N n)   = I n
evc (T t)   = B t
evc (Bao o) = F $ λa → F $ λb → I $ o (unIc a) (unIc b)
evc (Ipr p) = F (B ∘ p ∘ unIc)
evc Not     = F (B ∘ ¬ ∘ unBc)
evc If      = F $ λa → F $ λb → F $ λc →
                counit (if (unBc a) then b else c)

unIc  v = (unI  ∘ counit) v
unBc v = (unB ∘ counit) v
```

7 Identity comonadic semantics

```
instance Comonad Identity where
  counit (Identity a) = a
  cobind k = Identity ∘ k

instance Zippable Identity where zzipWith = lift2

instance DfComonad Identity   -- fbyg and nxtg left undefined

execIdentity k = k (Identity envEmpty)
```

8 Dataflow comonadic semantics

```
data HCF a = HCF{history :: [a], current :: a, future :: Stream a}

instance Comonad HCF where
  counit = current
  cobind k d = HCF (cobindL d) (k d) (cobindS d)
    where
    cobindL (HCF [] _ _)  = []
    cobindL (HCF (a' : az') a as) = k d' : cobindL d'
      where d' = HCF az' a' (a :< as)
    cobindS (HCF az a (a' :< as')) = k d' :< cobindS d'
      where d' = HCF (a : az) a' as'

instance Zippable HCF where
  zzipWith f (HCF az a as) (HCF bz b bs) =
    HCF (zipWith f az bz) (f a b) (lift2 f as bs)

instance DfComonad HCF where
  fbyg a0 (HCF [] _ _)  = a0
```

```
  fbyg _  (HCF (a :< _) _ _)          = a   -- extract latest history value
  nxtg    (HCF _        _ (a :< _))   = a   -- extract earliest future value
execHCF k = execHCF' k []
execHCF' k envEmpH = k env :< execHCF' k (envEmpty : envEmpH)
  where
  env = HCF envEmpH envEmpty envEmpF
envEmpF = envEmpty :< envEmpF
```

9 Generic AF-based semantics

```
class Applicative f where
  pure :: a → f a
  (⊛)  :: f (a → b) → f a → f b
lift1 :: Applicative f ⇒ (a → b)                → f a → f b
lift2 :: Applicative f ⇒ (a → b → c)            → f a → f b → f c
lift3 :: Applicative f ⇒ (a → b → c → d) → f a → f b → f c → f d
lift1 = (⊛)        ∘ pure   -- like map
lift2 = ((⊛)∘)     ∘ lift1  -- like zipWith
lift3 = (((⊛)∘)∘) ∘ lift2   -- like zipWith3
class Applicative g ⇒ DfApplicative g where
  fbyg :: g a → g a → g a
  nxtg :: g a → g a
data Val g = I {unI  :: g Integer}
           | B{unB   :: g Bool}
           | F{unF   :: Val g → Val g}
evMainAp a = ev a envEmpty
type En g = [(Var, Val g)]
ev :: DfApplicative g ⇒ Tm → En g → Val g
ev (V x)      = envLookup x
ev (L x a)    = λe → F $ λv → ev a (envUpdate x v e)
ev (A a b)    = s (unF ∘ (ev a)) (ev b)
ev (Y a)      = fix ∘ (unF ∘ (ev a))
ev (C c)      = const (evc c)
ev (Fby a b)  = s (fby ∘ (ev a)) (ev b)
ev Nxt        = const (F nxt)
evc (N n)   = I  (pure n)
evc (T t)   = B (pure t)
evc (Bao o) = F $ λa → F $ λb → I $ (lift2 o) (unI a) (unI b)
evc (Ipr p) = F (B ∘ lift1 p ∘ unI)
evc Not     = F (B ∘ lift1 ¬ ∘ unB)
evc If      = F $ λa → F $ λb → F $ λc →
              I $ (lift3 i) (unB a) (unI b) (unI c)
i a b c = if a then b else c
```

$fby\ (I\ \ a) \sim (I\ \ b) = I\ \ (fbyg\ a\ b)$
$fby\ (B\ \ a) \sim (B\ \ b) = B\ (fbyg\ a\ b)$
$nxt\ (I\ \ a) = I\ \ (nxtg\ a)$
$nxt\ (B\ a) = B\ (nxtg\ a)$

10 Identity AF semantics

instance *Applicative Identity* **where** $pure = return; (\circledast) = ap$
instance *DfApplicative Identity* -- *fbyg* and *nxtg* left undefined

11 Dataflow AF semantics

instance *Applicative Stream* **where**
$pure\ a = a :< pure\ a$
$\sim (f :< fs) \circledast \sim (a :< as) = f\ \ a :< (fs \circledast as)$
instance *DfApplicative Stream* **where**
$fbyg\ \ (a :< \_)\ \ = (a :<)$
$nxtg\ (\_ :< as) = as$

12 Interpreters

Aside from some instances of *Show* not shown, we now have 2 variants of 2 interpreters for *Df*. Given a closed term a, they can be invoked as follows.

$evMainAp\ a :: Val\ Identity$
$evMainAp\ a :: Val\ Stream$
$execIdentity\ (ev\ a)$
$execHCF\ \ \ (ev\ a)$

13 Conclusions

I have shown that it is simpler to separate the dataflow and non-dataflow parts of a language with an AF than it is with a comonad. With an AF, the core semantics shown below are the same as in a scalar language.

$ev\ (V\ x)\ \ = envLookup\ x$
$ev\ (L\ x\ a) = \lambda e \rightarrow F\ \$\ \lambda v \rightarrow ev\ a\ (envUpdate\ x\ v\ e)$
$ev\ (A\ a\ b) = s\ (unF \circ (ev\ a))\ (ev\ b)$

In contrast, note the need for *counit*, *zzipWith*, and *cobind* in the comonadic semantics below.

$ev\ (V\ x)\ \ = envLookup\ x \circ counit$
$ev\ (L\ x\ a) = \lambda e \rightarrow F\ \$\ \lambda v \rightarrow ev\ a\ (zzipWith\ (envUpdate\ x)\ v\ e)$
$ev\ (A\ a\ b) = s\ (unF \circ (ev\ a))\ (cobind\ (ev\ b))$

The AF approach is also simpler because making a dataflow-specific AF (making *Stream* an instance of *DfApplicative*) is simpler than making the corresponding

instance of a comonad (making *HCF* an instance of *DfComonad*). Note the greater size and complexity of Section 8 compared to Section 11.

With an AF, the semantics of the language's constants do look a little different than they would in a scalar language since they accept AF types. E.g., note the need for *pure* and *lift1* below.

$evc\ (T\ t) = B\ (pure\ t)$
$evc\ Not\ \ = F\ (B \circ lift1\ \neg \circ unB)$

In contrast, admittedly, the comonadic semantics of the ground constants are the same as in a scalar language. But, as in the AF semantics, the meanings of the function constants are a bit different. E.g., below, $evc\ (T\ t)$ is the same as in a scalar semantics, but note the need for *counit* (via *unBc*) in *evc Not*.

$evc\ (T\ t) = B\ t$
$evc\ Not = F\ (B \circ \neg \circ unBc)$
$unBc\ v = (unB \circ counit)\ v$

Finally, note that the dominant data type of the comonadic interpreter, *DfComonad*, is not just a *Comonad*. It also needs to be a *Zippable*. This weakens the argument of Uustalu and Vene by showing that the comonadic operations are insufficient. It also strengthens my argument, because the needed operation, a generalized version of *zipWith* I call *zzipWith*, is the same as the *lift2* operator you get for free with an *Applicative* (AF). In fact, the careful reader my have noticed that I have (generously, I think) simplified the implementations of *zzipWith* by using *lift2*, which relies on Identity and Stream already being instances of *Applicative*. Since you must already have something close to an AF (a *Zippable*) to implement a *DfComonad*, why not just stop there and use an AF-based semantics?

14 Future Work

I use AFs whereas Wadge uses monads [4] to describe the semantics of Lucid-like (intensional) languages. What is the relationship between these approaches? One possibly-big difference is Wadge addresses the semantics of **latest** whereas I do not: perhaps this is where monads are helpful?

References

[1] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20(7):519–526, 1977.

[2] Conor McBride and Ross Paterson. Applicative programming with effects. `http://www.cs.nott.ac.uk/~ctm/IdiomLite.pdf`.

[3] Tarmo Uustalu and Varmo Vene. The essence of dataflow programming. In *Lecture Notes in Computer Science*, volume 3780, pages 2–18, nov 2005.

[4] W. W. Wadge. Monads and intensionality. In *ISLIP '??*

[5] Philip Wadler. Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, New York, NY, USA, 1990. ACM Press.